ELEC 390 Principles of Design & Development:

Final Project Report

Presented to: Prof. Ali Etemad

Winter 2023

Faculty of Applied Science

Queen's University

Prepared by Group 40

Shams Sajid, 20217791, 19srs12@queensu.ca

Harshil Patel, 20250563, 20hp9@queensu.ca

Jasmine Klein, 20154586, 18jjdk@queensu.ca

Submitted on April 14, 2023

Data Collection

The data was collected using the “Acceleration with g” Raw Sensor provided by the mobile application, phyphox. Each team member tracked their “walking” and “jumping” movements for approximately five minutes at random to maximize diversity. After collecting the accelerometer data, it was exported to a CSV (Comma, decimal point) format which was labeled by name for each member then combined into a single CSV.

Challenges we faced during the data collection step occurred when performing jumping jacks. Sometimes the data would be off and it would not record properly which was made clear from the live output graph on phyphox. This was a challenge because we had to record it multiple times since the sensor was tracking additional movement from my hand moving up and down that did not coincide with vertical jumping. To overcome this challenge, it was decided to hold the mobile device against your person to improve accuracy.

Notable hardware used was iPads, iPhones ect… The softwares used were phyphox, and Microsoft Excel to load the CSV for analysis. Python libraries that were used in this project include tkinter, pandas, numpy, time, scipy.stats, scipy.signal (spectrogram), matplotlib, imblearn.over_sampling (SMOTE), joblib, keyboard, h5py, selenium, sklearn: StandardScalaer, GridSearchCV, LogisticRegression, train_test_split, accuracy_score, classification_report, confusion_matrix.

Data Storing

The collected data is stored in a hierarchical data format (HDF5) file, which allows for efficient storage and retrieval of large and complex datasets. The process of storing the collected data is described as follows.

Load the data from multiple CSV files and combine them into a single data frame using the `load_data` function. Each CSV file contains raw data for a specific person. One-hot encoding is applied to the 'Activity' column to create separate columns for each activity ('Activity Walking' and 'Activity Jumping'), with binary values indicating if the activity is taking place or not. If it is a value of 1 in the specific column, it indicates that the activity was performed.

Segment the combined data into 5-second windows using the `segment_data` function. This function takes the data frame, a window size (in seconds), and the sample rate (number of samples per second) as input arguments. The segmented data is a list of data frames, where each data frame represents a 5-second window of data.

Create train and test splits from the segmented data using the `create_splits` function. This function uses the `train_test_split` function from the `sklearn.model_selection` module to randomly shuffle the segments and split them into training and testing sets according to the specified test size (e.g., 10% for testing).

Process the metadata time information for each person using the `process_meta_data_time` function. This function calculates the duration and end time of each event, converts the system time to datetime format, and returns a data frame with the processed metadata.

Write the original data, train data, and test data to an HDF5 file using the `write_to_hdf5` function. This function creates an HDF5 file named `Combined_data.hdf5` and stores the data. The steps for this subprocess includes the following:

- For each person, a group is created with the person's name, and their data is stored as a dataset named 'data' within that group.
- A group named 'dataset' is created, which contains two subgroups: 'train' and 'test'. The train and test data frames are stored as datasets named `train_data` and `test_data` within their respective subgroups.
- Read the contents of the HDF5 file using the `reading_from_hdf5` function. This function reads the original data for each person, train data, and test data from the HDF5 file and returns them as separate data frames.

Visualization

The following section contains plots that were created to better understand and visualize samples from the dataset.

Acceleration vs Time Plots

Acceleration-time graphs were chosen to visualize the raw data picked up by the sensors. The plots contain the acceleration in the x, y and z directions with respect to time for each group member in the dataset. This visualization helps understand the general pattern of acceleration signals, any trends, or fluctuations over time, and how the acceleration components in different directions relate to each other.

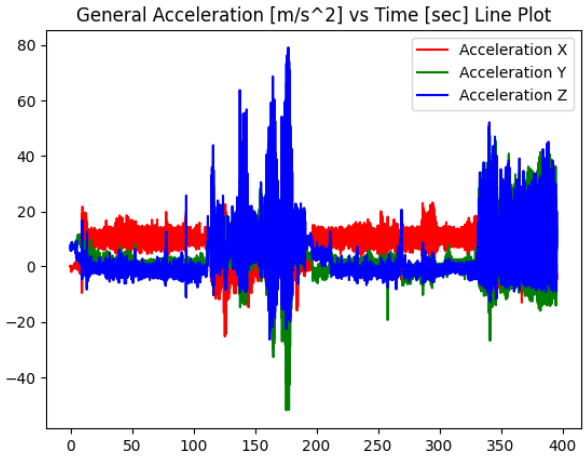


Figure 1: Acceleration of all three axes as a function of time for the raw data collected by Harshil.

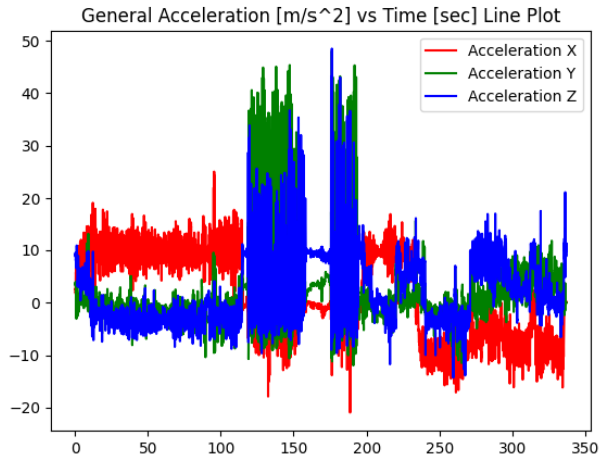


Figure 1: Acceleration of all three axes as a function of time for the raw data collected by Jasmine.

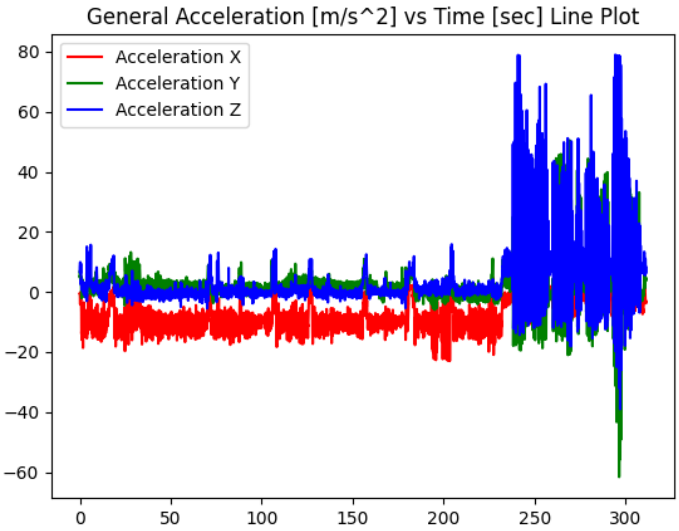


Figure 3: Acceleration of all three axes as a function of time for the raw data collected by Shams.

Metadata Time Gantt Chart and Sensor Activity Plots

These visualizations help analyze the duration and distribution of events or activities across different individuals in the study. By examining the Gantt chart, we identified overlaps, gaps, or patterns in the timing of events. The bar plot provides a summary of the total duration for each event or activity, allowing you to compare their relative frequency or importance in the dataset.

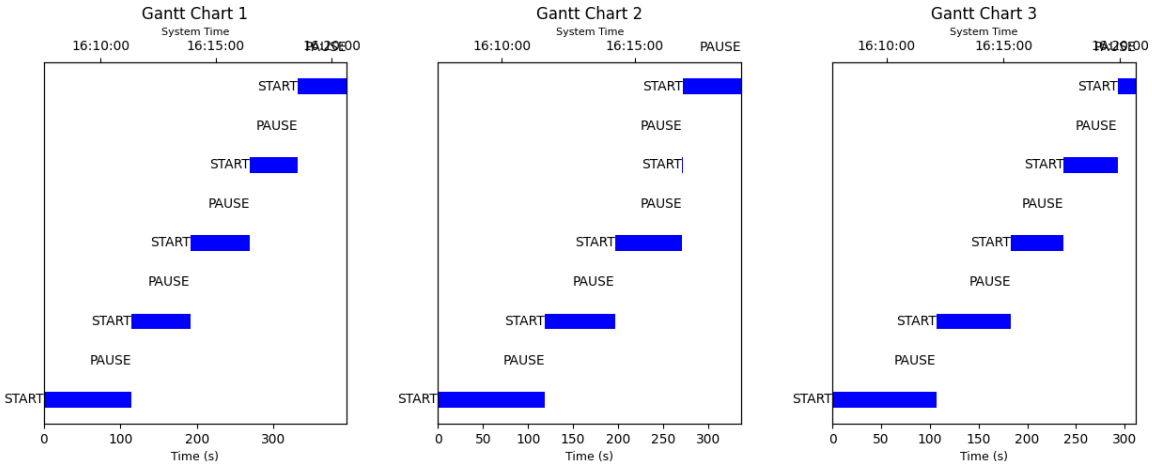


Figure 2: Then timeline of the sensor activity during the collection time displayed in a Gantt chart. The experiment continues to start and pause until the last recoreded event at time 311.253 seconds for each group member.

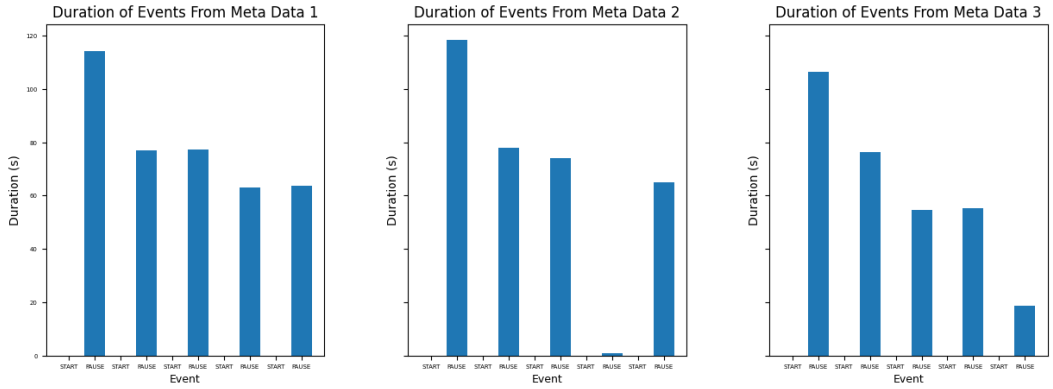


Figure 3: Visual analysis of the experiments meta-data. Each bar graph displays the duration of an event in the experiment for each group member.

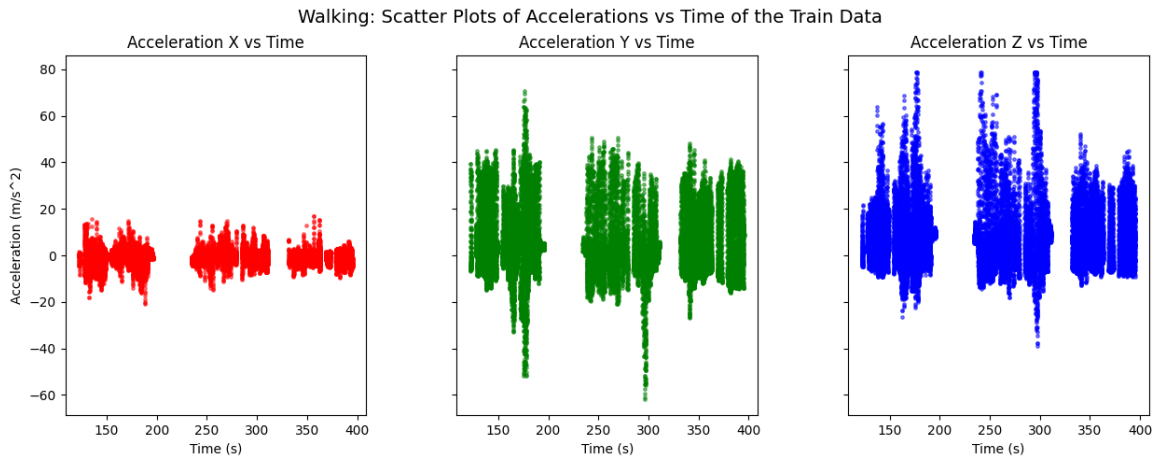


Figure 4: Acceleration of each axis displayed as a function of time for the pre-processing training set to classify walking.

Scatter Plots

Scatter plots of the walking and jumping activities for both the training and testing datasets help identify any relationships or trends between different variables, such as the acceleration components. These plots revealed any outliers or clusters in the data, which can also be used to indicate potential issues with the data collection or classification.

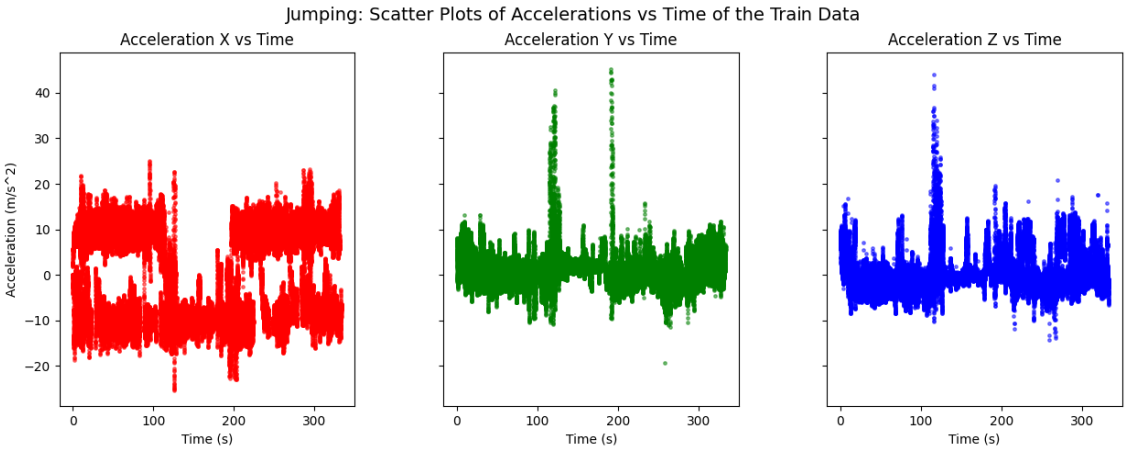


Figure 5: Acceleration of each axis displayed as a function of time for the pre-processing training set to classify jumping.

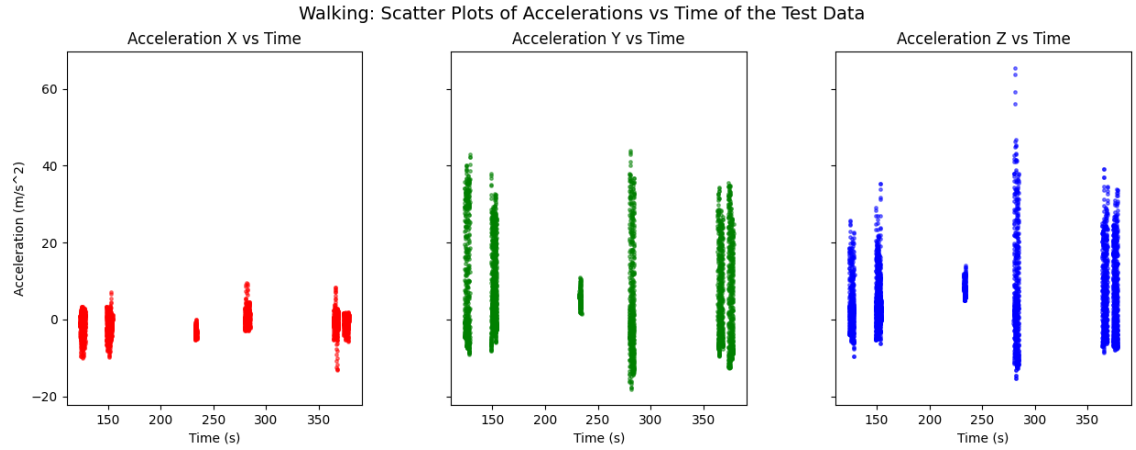


Figure 6: Acceleration of each axis displayed as a function of time for the test dataset (walking).

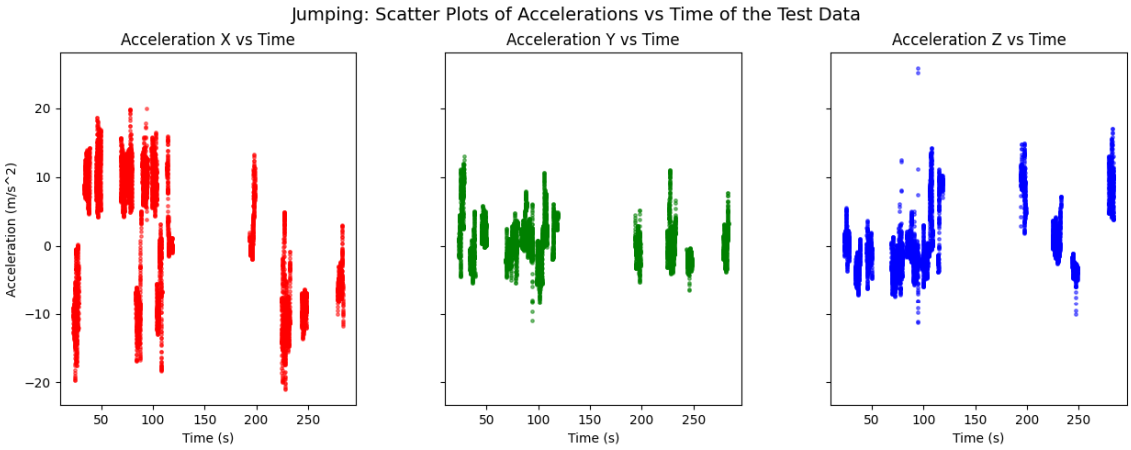


Figure 9: Acceleration of each axis displayed as a function of time for the test dataset (jumping).

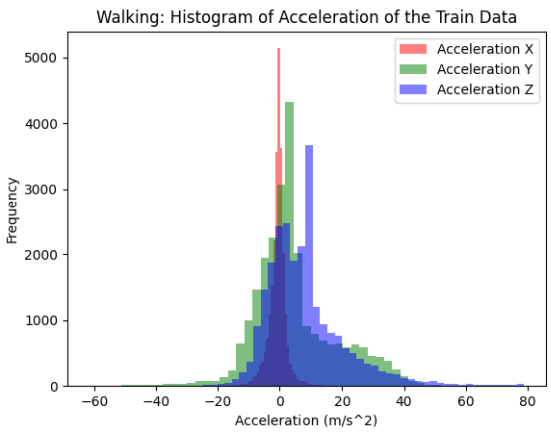


Figure 10: Histogram of the training data (walking) for all axes.

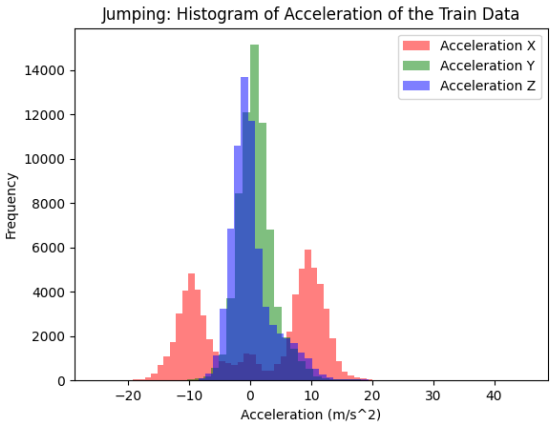


Figure 11: Histogram of the training data (jumping) for all axes.

Histograms

Histograms of the walking and jumping activities for both the training and testing datasets help visualize the distribution of the acceleration data. By examining these histograms, we identified the central tendency, spread, and shape of the data, as well as any potential skewness or kurtosis.

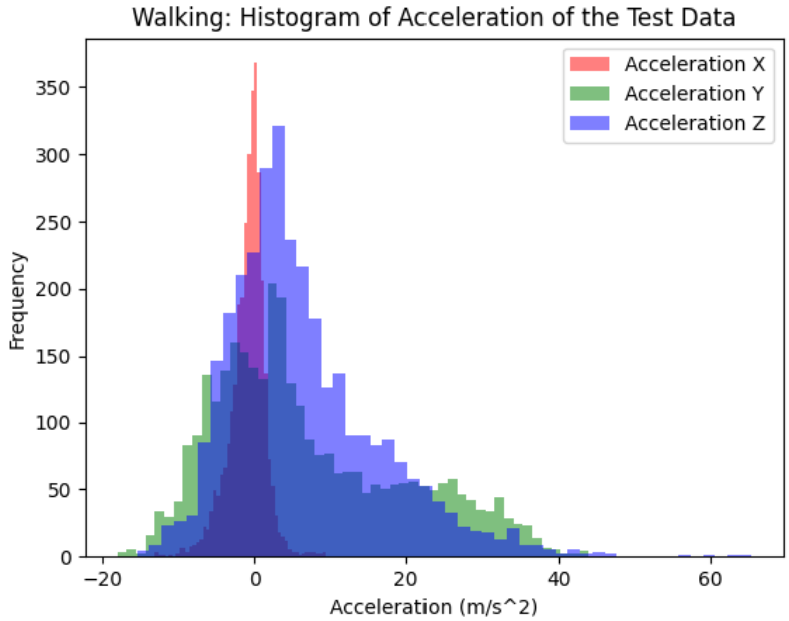


Figure 12: Histogram of the test data (walking) for all axes.

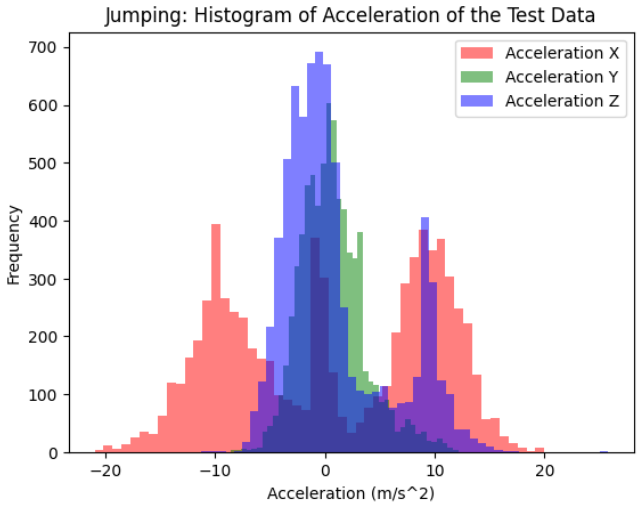


Figure 13: Histogram of the test data (jumping) for all axes.

Bar Plots of Mean

These plots show the mean values of the acceleration components for walking and jumping activities in both the training and testing datasets. By comparing the means, you can gain insights into the differences between the activities, which would be useful for developing or evaluating classification models.

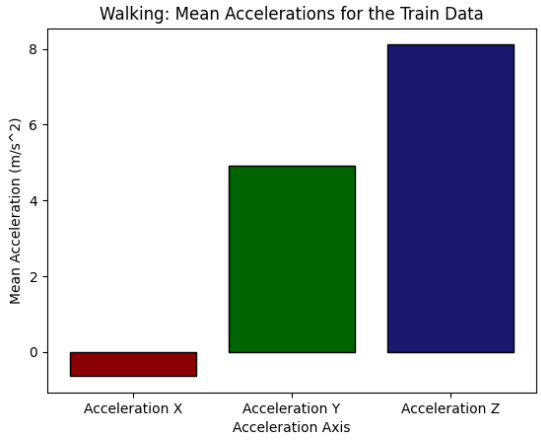


Figure 14: Extracted feature; mean acceleration of all axes for the training data (waking)

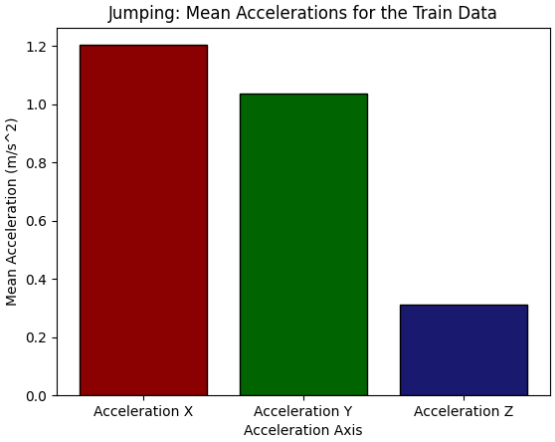


Figure 15: Extracted feature; mean acceleration of all axes for the training data (jumping).

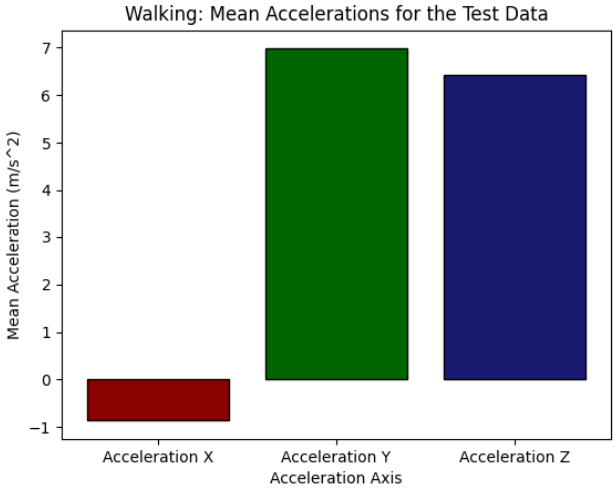


Figure 16: Extracted feature; mean acceleration of all axes for the test data (walking).

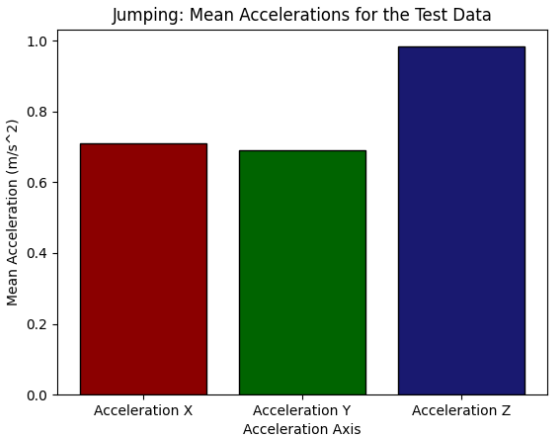


Figure 17: Extracted feature; mean acceleration of all axes for the test data (jumping).

Takeaways

The visual analysis of the complete dataset from the detailed plots above representing walking and jumping allowed for easy side-by-side comparison of each motion. This made it clear to fully understand the characteristics of acceleration and how it differs for each movement.

Using what was gained from the plots, improvements to the data collecting performance might be made to improve the entire dataset. Other sensors or data sources, such as a gyroscope or magnetometer, would be useful for gathering more information about the activities and improving categorization performance.

The visualization of the pre-processed data indicates that the data contains noise prior to normalization (visualizations provided below in the preprocessing section). Performing the data collection in a more controlled environment will reduce inconsistency in the data, this could include specifying the surface type, footwear or any other factors that might influence the quality of the data.

Consider lengthening the period of collection to guarantee that enough repetitions for each activity is captured for visualizations that show a shortage of data. The specification of activities, in conjunction with the metadata time bar plot, would ensure a balanced dataset. This is critical for developing a high-performing classification model and eliminating biases towards specific activities. Based on the visualization insights, it may be beneficial to integrate variations of the original motions (e.g., walking at different speeds, jumping with different approaches) to better comprehend the differences between the activities and improve the classification model's performance.

Pre-processing

In the preprocessing stage, several measures were taken to clean and prepare the data for further analysis. The data set was first examined for any missing or negligible values (ie. NaN and Null) and the total count of appearances were printed for each data file. Based on the presence of missing values, a method was chosen to handle them. In this case, the options were to either drop the rows containing missing values or to fill them with the mean of the respective columns.

Noise Reduction

To reduce noise in the data, a Moving Average (MA) filter was applied using different window sizes (5, 50, and 100). The MA filter acts as a low-pass filer which helped in smoothing the acceleration data and reducing short-term fluctuations. The choice of window sizes was made to explore the impact of different levels of smoothing on the data. The plots of the original and filtered data below show the effect of the MA filter.

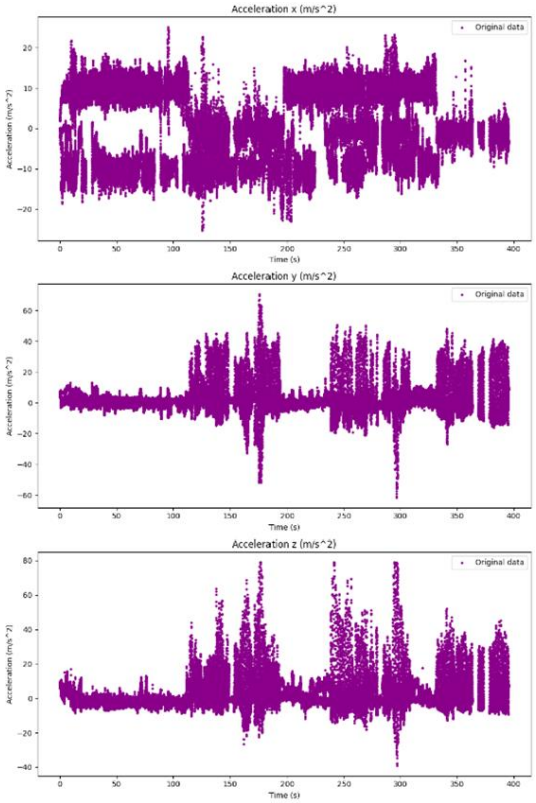


Figure 18: Scatter plot of the combined train acceleration values before applying the moving average filter.

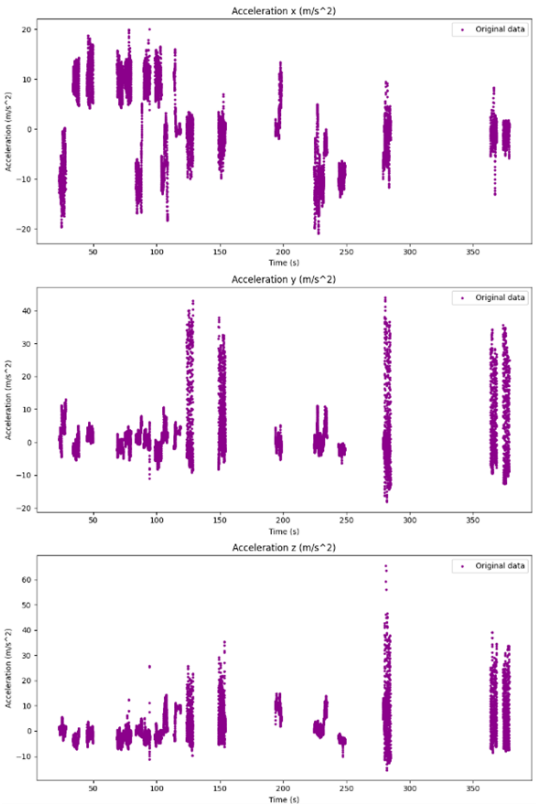


Figure 19: Scatter plot of the combined test acceleration values before applying the moving average filter.

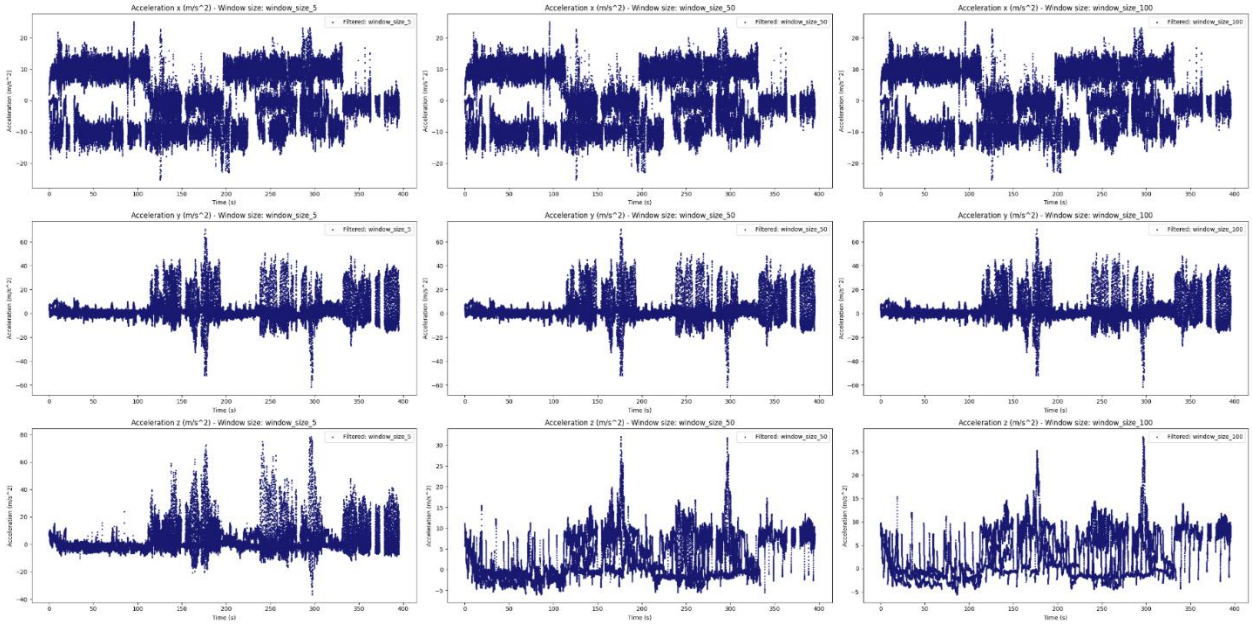


Figure 20: Acceleration values from the training data after applying a MA filter with window size 100.

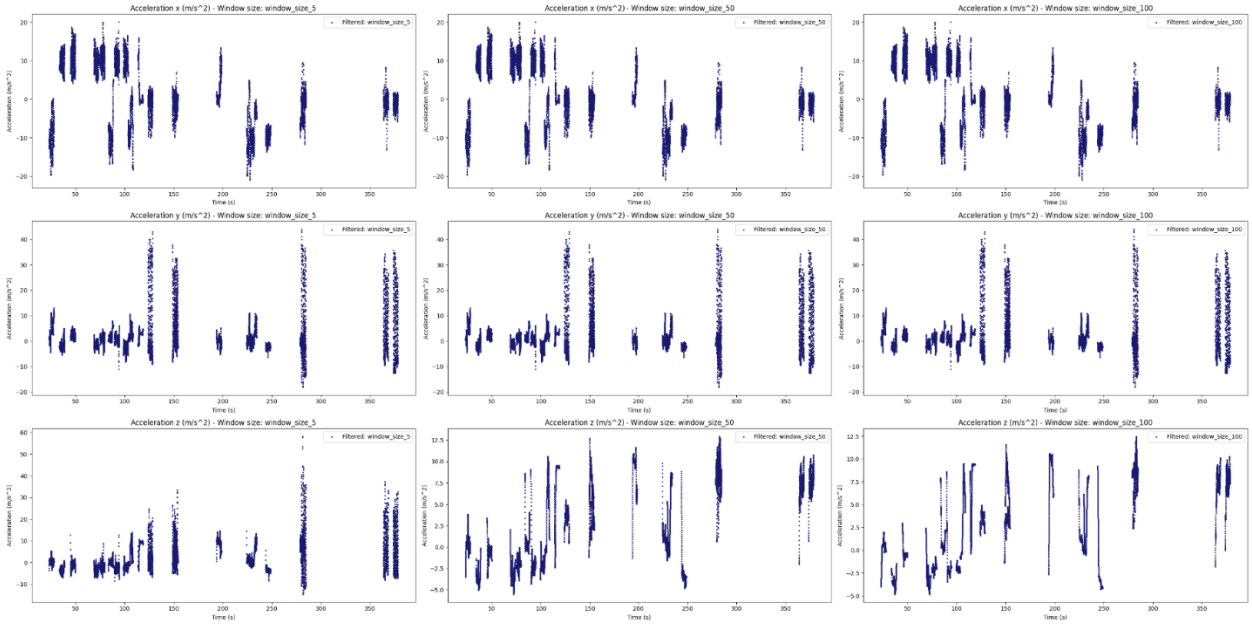


Figure 21: Acceleration values from the test data after applying a MA filter with window size 100.

Furthermore, an Exponential Moving Average (EMA) filter was applied to the data for the purposes of noise reduction. An EMA filter is a low-pass filter that places more weight on the recent data by discontinuing old data in exponential fashion [6]. Based on the mathematical model of an EMA filter shown below, the value of 0.1 was chosen for the constant alpha. This

value was selected to provide a balance between smoothing and responsiveness to changes in the data.

$$y[i] = \alpha * x[i] + (1 - \alpha) * y[i - 1] \quad (1)$$

The constant alpha determines how aggressive the filter is by setting the cut off frequency where $\alpha \in [0,1]$.

The parameters, such as the window sizes for the moving average filter and the alpha for the exponential moving average filter, were chosen to experiment with varying levels of smoothness and responsiveness to changes in the data. The moving average filter was designed with a 100-window size to reduce noise in the data by averaging over a larger number of data points. As a result, the signal becomes smoother, making it easier to discover underlying patterns and trends in the data.

The parameters, such as the window sizes for the moving average filter and the alpha for the exponential moving average filter, were chosen to experiment with varying levels of smoothness and responsiveness to changes in the data. The moving average filter was designed with a 100-window size to reduce noise in the data by averaging over a larger number of data points. As a result, the signal becomes smoother, making it easier to discover underlying patterns and trends in the data.

Outlier Detection and Removal

The Interquartile Range (IQR) was used help identify and remove potential outliers. The IQR method was set up with a threshold of 2, this helped with the elimination of extreme data points that could negatively impact the analysis. The IQR is computed by finding the median followed by Q1 and Q3 of our ordered dataset as shown below [7].

$$2 * IQR = 2 * (Q1 - Q3) \quad (2)$$

Removal of outliers caused an imbalance in the dataset. To handle the imbalance the dataset, the Sythentic Minority Over-sampling Technique (SMOTE) was implemented by resampling the minority class. The value of four was chosen for the number nearest neighbours as a reasonable balance between oversampling complexity and maintain the original distribution.

The influence of amplification factors and noise recorded by the accelerometer sensor negatively affects the visual representation the of data collected. To avoid inaccurate patterns, the dataset was made suitable by modifying the data such that it has a zero mean and fits a particular range. This performance of normalization on the acceleration data was done using the StandardScaler from the scikit-learn library. This step is essential for regression models due to the scale sensitivity of the input features; namely, logistic regression which produces a probability value.

Histograms of the normalized data were plotted to visually represent the distribution of the acceleration data across the x, y and z axis which can be seen below.

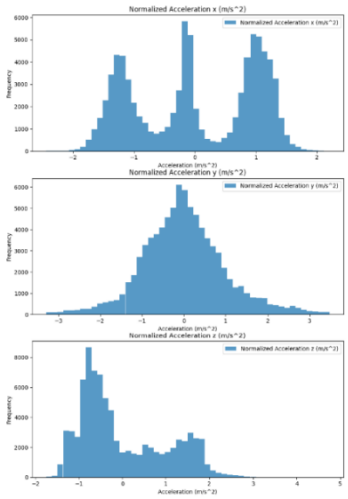


Figure 7: Histogram of the normalized data for each axis.

Overall, the pre-processing measures taken had several impacts on the dataset with the intention to make the original accelerometer data more suitable for analysis and model training. This includes noise reduction done by applying MA and EMA filters making it optimal for visual inspection, the removal of potential outliers for better consistency and robust to extreme values, addressing issues of class imbalance with the application of SMOTE to ensure that the classification models are trained on a balanced set and finally normalization of the entire dataset done by scaling the features to a common scale.

Feature Extraction

Below are the features that were used in the project's coding.

Table 1: Extracted features of the dataset and the description of the significance

Extracted Feature	**Significance**
Max	The segment's highest possible value. It provides details regarding the highest acceleration peak.
Min	The segment's lowest value. It provides details regarding the lowest acceleration value.
Range	The distance between the highest and smallest values, which indicates how evenly distributed the data is.
Mean	The segment's average value of data, which has a general central tendency.

Median	The middle value when the data is sorted. It is less sensitive to outliers than the mean.
Variance	The average of the squared differences from the mean. It measures the dispersion of the data.
Standard Deviation	The square root of the variance, giving a measure of dispersion that is in the same unit as the data.
Skewness	A measure of the asymmetry of the probability distribution of the data around its mean. It indicates if the data is skewed to the left or right.
Kurtosis	A measure of the "tailedness" of the data distribution. It indicates the presence of outliers or extreme values.
Root Mean Square	The square root of the mean of the squared values. It provides an overall magnitude of the acceleration signal.
Mean Absolute Change	The average of the absolute differences between consecutive data points, representing the average change in the signal.
Mean Crossing Rate	The average rate at which the data crosses its mean value. It can be used to identify rhythmic patterns or periodicity in the data.

These characteristics were chosen because they can distinguish between different activities by capturing a wide range of qualities from accelerometer data. These features are often used in time-series analysis and have been proved in multiple studies to be successful for human activity recognition (HAR) tasks. These characteristics accurately describe the acceleration signals' core tendency, variability, shape, and temporal structure.

Numerous studies and methodologies in the literature (references are supplied in the appendix) have demonstrated how effectively these qualities perform when recognising distinct human activities with mobile sensors. The goal was to create a consistent and accurate activity recognition model that could distinguish between different activities, such as walking and leaping, based on the characteristics of the acceleration signals associated with each action.

The process of extracting these features involves segmenting the accelerometer data into smaller windows and calculating the desired features for each window. This approach helps reduce the dimensionality of the data and focuses on the most informative aspects of the signal, making it more suitable for machine learning models. By extracting these features, you can create a more compact and meaningful representation of the raw accelerometer data that can be used to train and evaluate classifiers for activity recognition.

Training the classifier

This section describes the process of training the logistic regression model is done in the `data_classifier.py` module.

The code defines a function called `train_and_evaluate_logistic_regression` that trains and evaluates a logistic regression model for the given acceleration data. The function accepts four arguments; namely, `train_data_normalized_y` and `test_data_normalized_y` which are the target variables for the training and testing datasets as well as `train_data_features` and `test_data_features` which are the feature matrices for the training and testing datasets.

The function starts by splitting the training dataset into training and validation sets using the `train_test_split` method from the `sklearn.model_selection` imported module. Then, it creates a logistic regression model with the `LogisticRegression` class from the `sklearn.linear_model` module. The model is configured with a maximum number of iterations, class weights set to 'balanced', and a random state for reproducibility.

Next, a dictionary of hyperparameters is created for tuning the model. The `GridSearchCV` class from the `sklearn.model_selection` module is used to find the best hyperparameters for the logistic regression model by performing a 5-fold cross-validation.

After obtaining the best model, it is evaluated on the validation set. The performance metrics include accuracy, classification report, and confusion matrix. The function then proceeds to evaluate the best model on the test dataset using the same performance metrics which returns the best logistic regression model.

```
Validation Accuracy: 0.8148148148148148
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.80      0.80        25
           1       0.83      0.83      0.83        29

    accuracy                           0.81        54
   macro avg       0.81      0.81      0.81        54
weighted avg       0.81      0.81      0.81        54

Confusion Matrix:
 [[20  5]
 [ 5 24]]
Test Accuracy: 0.8888888888888888
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.80      0.80         5
           1       0.92      0.92      0.92        13

    accuracy                           0.89        18
   macro avg       0.86      0.86      0.86        18
weighted avg       0.89      0.89      0.89        18

Confusion Matrix:
 [[ 4  1]
 [ 1 12]]
```

Figure 8:Output of the classification report, confusion matrix and validation accuracy for both motions.

The parameters described below were used in the `train_and_evaluate_logistic_regression` function. These parameter choices help in finding a logistic regression model that best fits the data while avoiding overfitting and handling imbalanced datasets.

Table 2: The parameters used in the function responsible for training the classifier with their corresponding set values and description/justification.

Parameter	Value	Description
`max_iter`	1000	The maximum number of iterations for the logistic regression model to converge. A higher value is chosen to ensure that the model has enough iterations to find the optimal solution, especially if the data is complex.
`class_weight`	'balanced'	Adjusts the weights of the classes based on their frequencies in the input data. This is useful when dealing with imbalanced datasets, as it helps to prevent the model from being biased towards the majority class.
`random_state`	42	A random seed value that ensures the results are reproducible. By setting a fixed random state, the same results can be obtained each time the model is trained.
`C`	A range of values [0.01, 0.1, 1, 10, 100, 1000] is provided for `GridSearchCV` to find the best value. Smaller values prevent overfitting, while larger values allow the model to fit more complex relationships in the data.	The C parameter in the `param_grid` dictionary is a regularization parameter for the logistic regression model. Smaller values of C create a stronger regularization effect, while larger values reduce the regularization effect.
`solver`	A list of solvers ['newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'] is provided for `GridSearchCV` to find the best one for the given dataset based on the specific characteristics of the dataset.	The solver parameter in the `param_grid` dictionary specifies the optimization algorithm to be used for training the logistic regression model.

Model deployment

The GUI design is depicted in the images above. The GUI was created in pychram using tkinter and designed in Python. The GUI's home page is a simple page with three buttons to pick from and a great background. The chosen background was a sports background, which was ideal for the project because the purpose was to determine whether the subject was walking or jumping. The color chosen was blue since it depicts summer and is where many of the activities take place.

When the instruction button is clicked, a new window will open with the instructions for properly usage of the software, making it easy for anyone to set up the project and utilize. To show how the GUI works, a number of programs had to be run to provide training data.

First, the main program was executed, which generated the test data required for the test. The primary component of the project is displayed after clicking the experiment data button. When that button is pressed, the user will be prompted to choose the location where the collected data is stored, this also runs the classier function. Once that file is chosen, a second window will appear indicating whether the individual is walking or jumping.

Finally, there is a button that displays real-time data. This button will take the user to phyphox, where they can perform a series of tests involving walking and jumping. After a while, the console window will display whether the user was walking or jumping, as well as the exact moment when this activity was completed. A detailed look at these buttons can be found in the team's demo video. The video demonstrates how the program works, while another video shows real-time statistics.

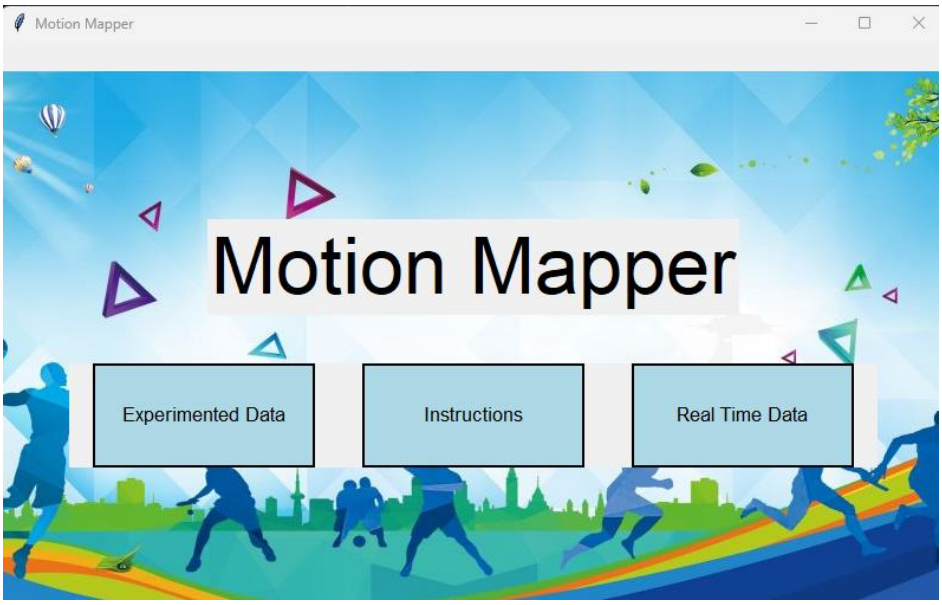


Figure 9: Initial window of the GUI when you first run it. Three options appear as buttons that are representative of the different analysis

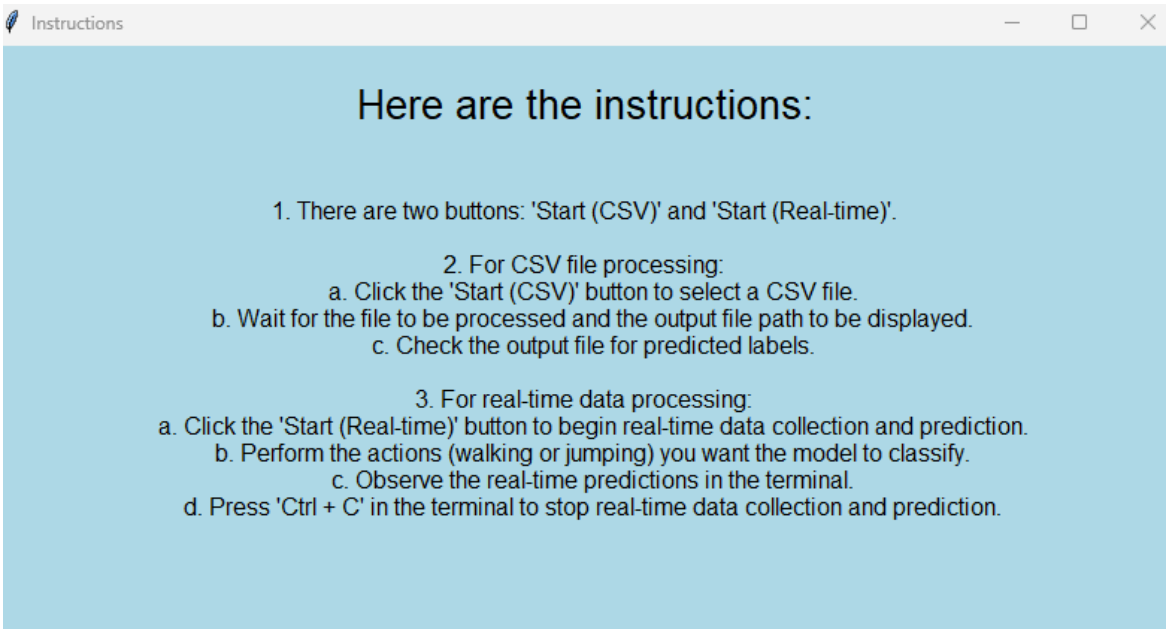


Figure 10: Instruction window on how to use the GUI

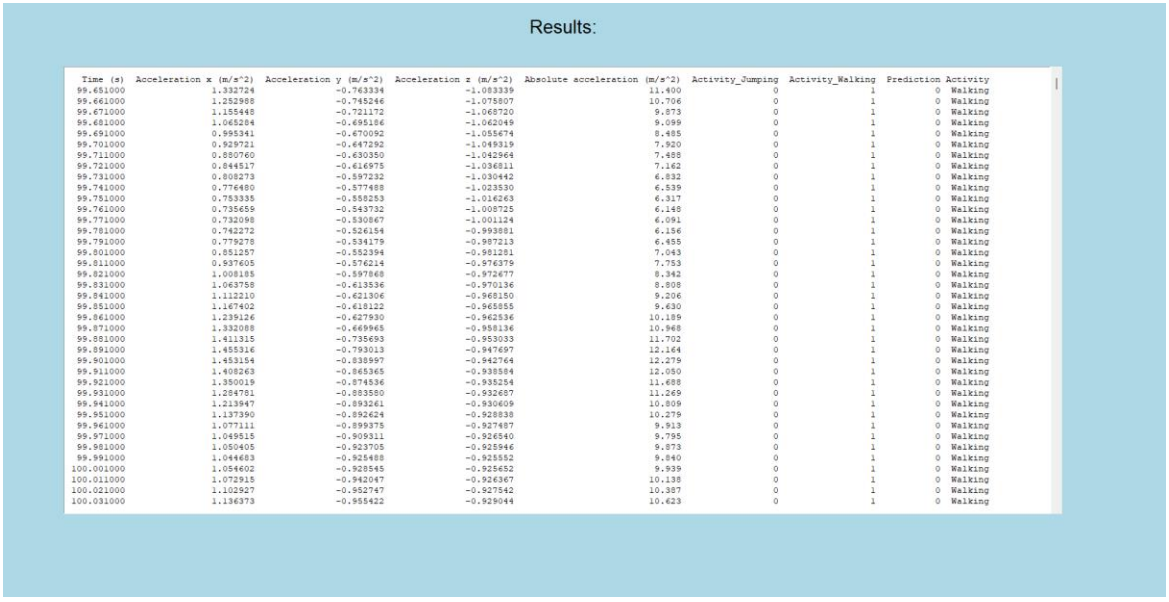


Figure 11: Output displayed on GUI window

Bonus Section

As previously mentioned, GUI has three button options upon the initial window as seen in Figure 28. For collection of the Real-Time and being able to predict real time activity, the user is required to click on the button in which he is asked to enter his remote access url from the Phyphox app.

The real-time data collection process is done using the following steps:

- Initialize the driver: The `setup_driver` function creates a new Chrome WebDriver instance and navigates to the specified URL. The URL is the location where the Phyphox application streams accelerometer data from a smartphone. The function then selects "Simple" mode, which simplifies the data display.

- Initial wait: The `real_time_classification` function begins by initializing the driver and waiting for a specified amount of time (the default is 3 seconds). This gives the data gathering time to settle before beginning the real-time analysis.

- Data collection: The function creates an empty DataFrame to contain the data collected. It then starts a loop, which repeats until the appropriate number of data points (`data_points`) are collected.

- Fetch data from Phyphox: The `fetch_data_from_phyphox` function is used within the loop to retrieve accelerometer data from the smartphone via the WebDriver. This function checks the webpage for the presence of acceleration elements and retrieves the x, y, and z acceleration values, as well as the absolute acceleration.

- Fetched data attached: The fetched data is turned into a DataFrame and appended to the main DataFrame containing the collected data. The DataFrame additionally calculates and stores the relative time since the start of data gathering.

- Real-time processing: Once a sufficient number of initial data points (`initial_data_points)` have been collected, the obtained data is analyzed in real-time. For analysis, the data is cleaned, filtered with moving average and exponential moving average filters, normalized, and divided into smaller windows.

- Feature extraction and prediction: From the segmented data, features are retrieved, and the trained machine learning model (`log_reg_model`) is utilized to predict activity labels for the data segments. The most recent prediction and time are displayed on the terminal.

- Sleep interval: Before obtaining the next set of data points, the function sleeps for a defined interval (`sleep_interval)` which allows you to regulate the frequency of data collecting.

- Exit conditions: The loop continues until the desired number of data points is collected, the escape key is pressed, or a KeyboardInterrupt (Ctrl+C) is issued. Once the loop ends, the WebDriver is closed.

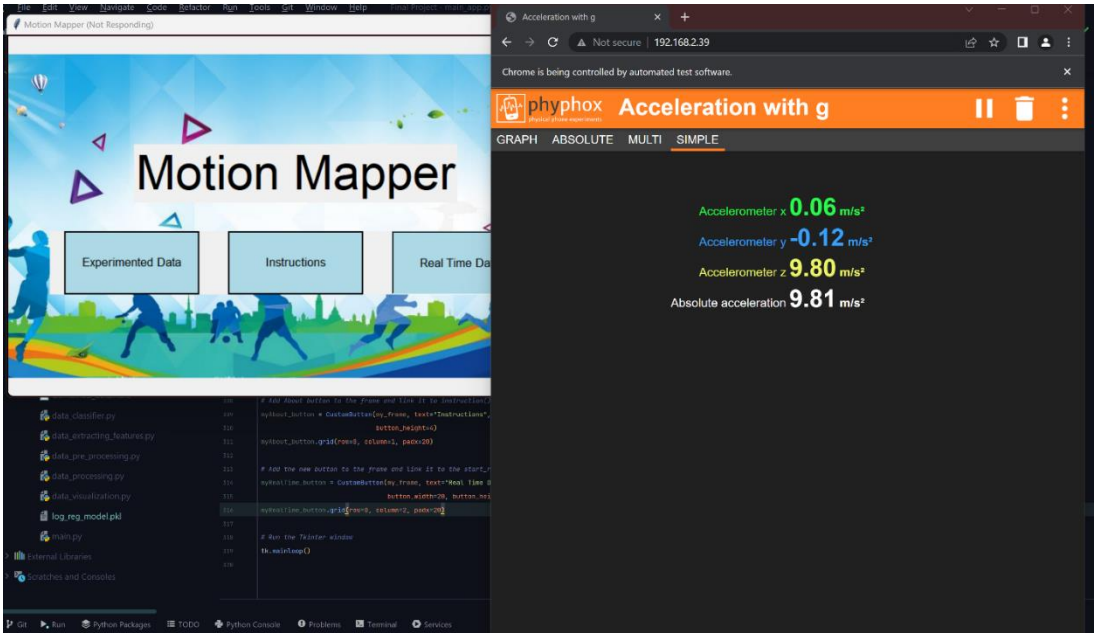


Figure 12: Side-by -side real time data collection

```
C:\Users\User\AppData\Local\Programs\Python\Python311\python
Time: 39.29s | Latest prediction: Jumping
Time: 39.41s | Latest prediction: Jumping
Time: 39.51s | Latest prediction: Jumping
Time: 39.61s | Latest prediction: Jumping
Time: 39.71s | Latest prediction: Jumping
Time: 39.81s | Latest prediction: Jumping
Time: 39.92s | Latest prediction: Jumping
Time: 40.02s | Latest prediction: Jumping
Time: 40.14s | Latest prediction: Jumping
Time: 40.25s | Latest prediction: Jumping
```

Figure 13: Real time data collection output for the jumping motion

The output shows the results of real-time activity predictions using the trained model which can be seen above in Figure 32. The script collects accelerometer data from a smartphone, processes it, and feeds it into the model to predict the activity being performed. In this case, the activity is "Jumping."

Each line of the output represents a prediction at a specific time:

- Time: The time (in seconds) elapsed since the beginning of data collection. It is shown with two decimal places for higher precision.
- Latest prediction: The predicted activity at the corresponding time.

There is a bit of a delay when getting the results for the output. This delay occurs since it takes time for python to first train the model, perform the prediction and then print out the results infinitely until paused. Also, the results are not completely accurate since more parameters are required to give the model enough time to distinguish the difference between walking and jumping data in real time.

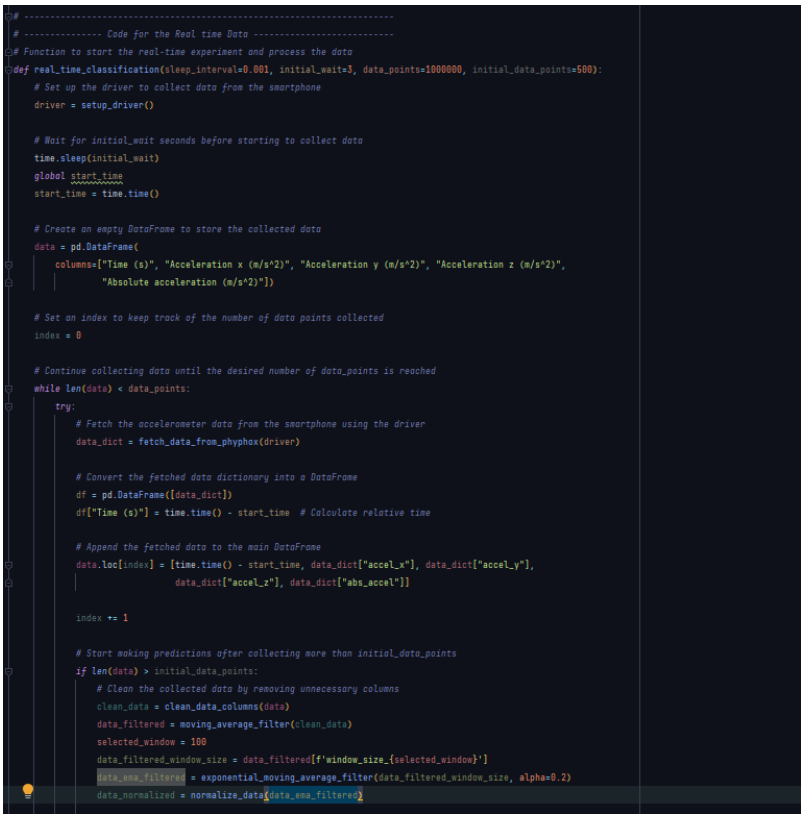
Bonus code

```
# This code will take the real time data from the Phyphox URL and then give the predictions in real time

# Importing the necessary libraries
import time
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

# Enter URL here
url = input("Please enter the Phyphox URL: ")


def setup_driver():
    # Initialize the Chrome WebDriver and open the provided URL
    driver = webdriver.Chrome()
    driver.get(url)

    # Click on the "Simple" mode
    driver.find_element(By.XPATH, "//li[text()='Simple']").click()

    return driver
```

Figure 14: Code snippet for collection of the real time data.

```python
def fetch_data_from_phyphox(driver):
    # Wait for the presence of the acceleration elements and fetch them
    accel_x_element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "element10")))
    accel_y_element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "element11")))
    accel_z_element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "element12")))
    abs_accel_element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, "element13")))

    # Extract acceleration data for each axis
    accel_x = extract_accel_data(accel_x_element)
    accel_y = extract_accel_data(accel_y_element)
    accel_z = extract_accel_data(accel_z_element)
    abs_accel = extract_accel_data(abs_accel_element)

    # Create a dictionary containing the time and acceleration data
    data = {
        "time": time.time(),
        "accel_x": accel_x,
        "accel_y": accel_y,
        "accel_z": accel_z,
        "abs_accel": abs_accel,
    }

    return data


def extract_accel_data(element):
    # Find the value number element within the provided element
    value_number = element.find_element(By.CSS_SELECTOR, ".value > .valueNumber")
    value_text = value_number.text

    # If the value text is empty, return 0.0
    if value_text == "":
        return 0.0

    # Convert the value text to a float and return it
    data_value = float(value_text)
    return data_value
```

Figure 15: Code snippet for collection of the real time data.



```
# ------------------------------------------------------------------
# -------------- Code for the Real time Data --------------------------
# Function to start the real-time experiment and process the data
def real_time_classification(sleep_interval=0.001, initial_wait=3, data_points=1000000, initial_data_points=500):
    # Set up the driver to collect data from the smartphone
    driver = setup_driver()

    # Wait for initial_wait seconds before starting to collect data
    time.sleep(initial_wait)
    global start_time
    start_time = time.time()

    # Create an empty DataFrame to store the collected data
    data = pd.DataFrame(
        columns=["Time (s)", "Acceleration x (m/s^2)", "Acceleration y (m/s^2)", "Acceleration z (m/s^2)",
                 "Absolute acceleration (m/s^2)"])

    # Set an index to keep track of the number of data points collected
    index = 0

    # Continue collecting data until the desired number of data_points is reached
    while len(data) < data_points:
        try:
            # Fetch the accelerometer data from the smartphone using the driver
            data_dict = fetch_data_from_phyphox(driver)

            # Convert the fetched data dictionary into a DataFrame
            df = pd.DataFrame([data_dict])
            df["Time (s)"] = time.time() - start_time  # Calculate relative time

            # Append the fetched data to the main DataFrame
            data.loc[index] = [time.time() - start_time, data_dict["accel_x"], data_dict["accel_y"],
                               data_dict["accel_z"], data_dict["abs_accel"]]

            index += 1

            # Start making predictions after collecting more than initial_data_points
            if len(data) > initial_data_points:
                # Clean the collected data by removing unnecessary columns
                clean_data = clean_data_columns(data)
                data_filtered = moving_average_filter(clean_data)
                selected_window = 100
                data_filtered_window_size = data_filtered[f'window_size_{selected_window}']
                data_ema_filtered = exponential_moving_average_filter(data_filtered_window_size, alpha=0.2)
                data_normalized = normalize_data(data_ema_filtered)
```

Figure 31: Code snippet for processing the real time data collection in the app.

```
                # Segment the cleaned data into smaller windows for analysis
                segmented_data = convert_to_segments(data_normalized, window_size=1, sample_rate=100)

                # Extract features from the segmented data to be used for classification
                features = extract_features(segmented_data)

                # Predict labels for the data segments using the trained model
                if not features.empty:
                    prediction = log_reg_model.predict(features)

                    # Print the latest prediction and time to the terminal
                    latest_time = data["Time (s)"].iloc[-1]
                    if prediction[-1] == 0:
                        print(f"Time: {latest_time:.2f}s | Latest prediction: Walking")
                    elif prediction[-1] == 1:
                        print(f"Time: {latest_time:.2f}s | Latest prediction: Jumping")
                else:
                    print("No features extracted. Please check the segmentation and feature extraction steps.")

            # Sleep for the specified interval before fetching the data again
            time.sleep(sleep_interval)

            # Check if the escape key has been pressed, and exit the loop if it has
            if keyboard.is_pressed("esc"):
                break

        except KeyboardInterrupt:
            # Break the loop when the user presses Ctrl+C
            break

    # Close the driver when the loop ends
    driver.quit()
```

Figure 32: Code snippet for processing the real time data collection in the app.

References

[1] J. R. Kwapisz, G. M. Weiss, and S. A. Moore, "Activity recognition using cell phone accelerometers," ACM SigKDD Explorations Newsletter, vol. 12, no. 2, pp. 74-82, 2011. [Online]. Available: https://dl.acm.org/doi/10.1145/1964897.1964918

[2] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. L. Reyes-Ortiz, "A public domain dataset for human activity recognition using smartphones," in ESANN, 2013, pp. 437-446. [Online]. Available: https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2013-84.pdf

[3] L. **Invalid source specified.** L. Zhang, X. Wu, and D. Luo, "Recognizing human activities from raw accelerometer data using deep neural networks," in 2016 IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom), 2016, pp. 1-6. [Online]. Available: https://ieeexplore.ieee.org/document/7749515

[4].Invalid source specified. M. Zeng, L. T. Nguyen, B. Yu, O. J. Mengshoel, J. Zhu, P. Wu, and J. Zhang, "Convolutional neural networks for human activity recognition using mobile sensors," in 6th International Conference on Mobile Computing, Applications and Services, 2014, pp. 197-205. [Online]. Available: https://ieeexplore.ieee.org/document/7026292

[5] L. Bao and S. S. Intille, "Activity recognition from user-annotated acceleration data," in International Conference on Pervasive Computing, Springer, Berlin, Heidelberg, 2004, pp. 1-17. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-24646-6_1

[6] mbedded.ninja. "Exponential Moving Average (EMA) Filters." [Online]. Available: https://blog.mbedded.ninja/programming/algorithms/exponential-moving-average-ema-filter/. [Accessed: Apr. 14, 2023].

[7] Statistics By Jim. **Invalid source specified.**Statistics By Jim. "Interquartile Range (IQR): How to Find and Use It." [Online]. Available: https://statisticsbyjim.com/basics/measures-of-spread-interquartile-range/. [Accessed: Apr. 14, 2023].

Participation Report

Task	Contribution
Data Collection	1/3, 1/3, 1/.3
Technical Work	1/3, 1/3, 1/3
Written Report	1/3, 1/3, 1/3